



Data Structures and Algorithms

Estruturas de Dados e Algoritmos

Capítulo 5

Estruturas Dinâmicas de Dados com Listas Ligadas

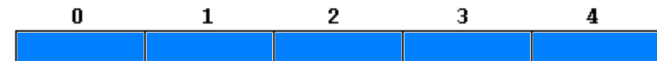


Estruturas Dinâmicas de Dados com Listas Ligadas

- Listas Ligadas (1ª e 2ª aulas)
- Outros tipos de listas ligadas (2ª aula)
- Pilhas (3ª aula)
- Filas (3ª aula)
- Exemplos com Pilhas (3ª aula)
- Exemplos adicionais (final semestre)



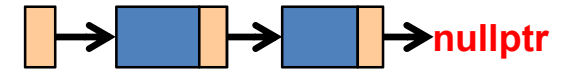
Vetores



- Vantagens:
 - Acesso muito rápido através do índice do elemento.
 - Programação simples e intuitiva.
- Desvantagens:
 - Tamanho máximo fixo.
 - Alocação de memória estática numa zona contígua.
 - Inserção ou remoção de elementos no meio do vetor pode implicar a movimentação de um número elevado de elementos.
 - Se o tamanho do vetor aumentar frequentemente, é necessário redimensionar a sua capacidade, alocando uma nova zona de memória contígua e copiando dados de um lado para outro.






Listas Ligadas



- Permitem armazenar em memória dados do mesmo tipo de forma **mais flexível** do que os vetores.
- A alocação de **memória** é **dinâmica**, em *run-time*.
- O seu **tamanho** – o número de nós – **pode crescer** (ou diminuir) dinamicamente.
- **Inserções e remoções** não implicam deslocamento de elementos; apenas manipulação de ponteiros.



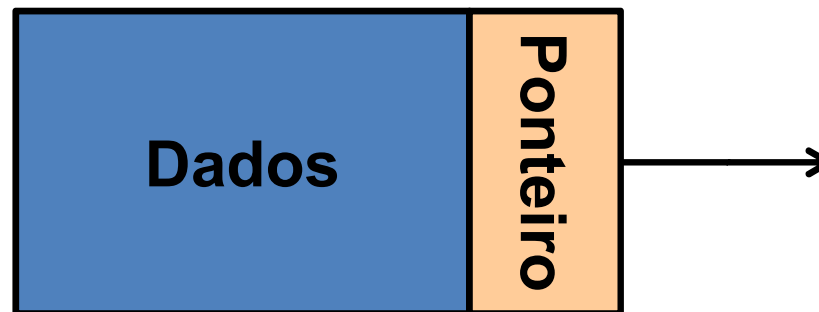
Listas Ligadas

- Constituídas por conjuntos de **nós** que armazenam os dados pretendidos.
- Cada nó armazena uma variável do tipo de dados da lista e um **ponteiro para o próximo nó**.

 - A lista pode ser duplamente ligada se armazenar também um ponteiro para o nó anterior.

- Para referenciar o **primeiro elemento** da lista utiliza-se um ponteiro que se designa por **cabeça da lista**.

 - Se a lista for duplamente ligada, utiliza-se também um ponteiro para referenciar o **último elemento** da lista que se designa por **cauda da lista**.



Lista Ligada

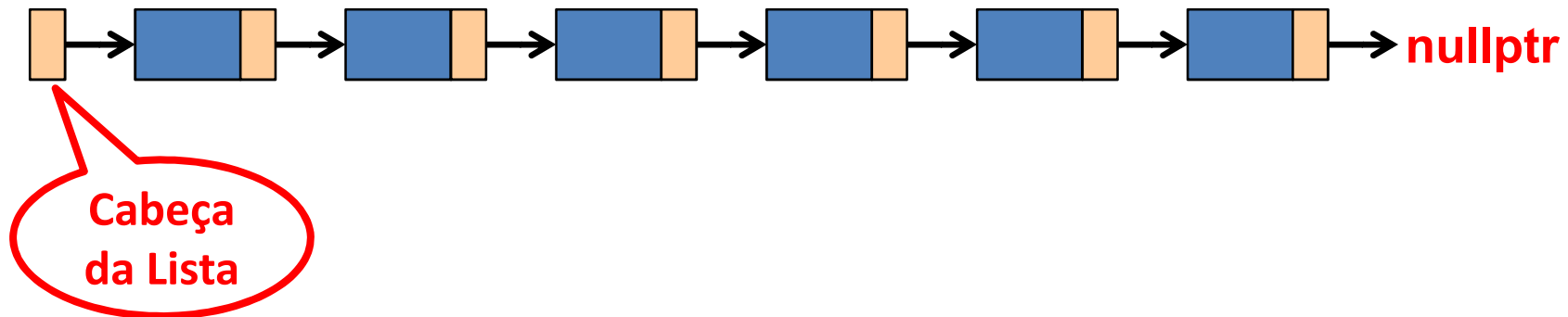
- Composição de cada **nó**:
 - Dados: estrutura de dados que permite armazenar o tipo de dados da lista.
 - Ponteiro: aponta para o próximo nó da lista.





Lista Ligada

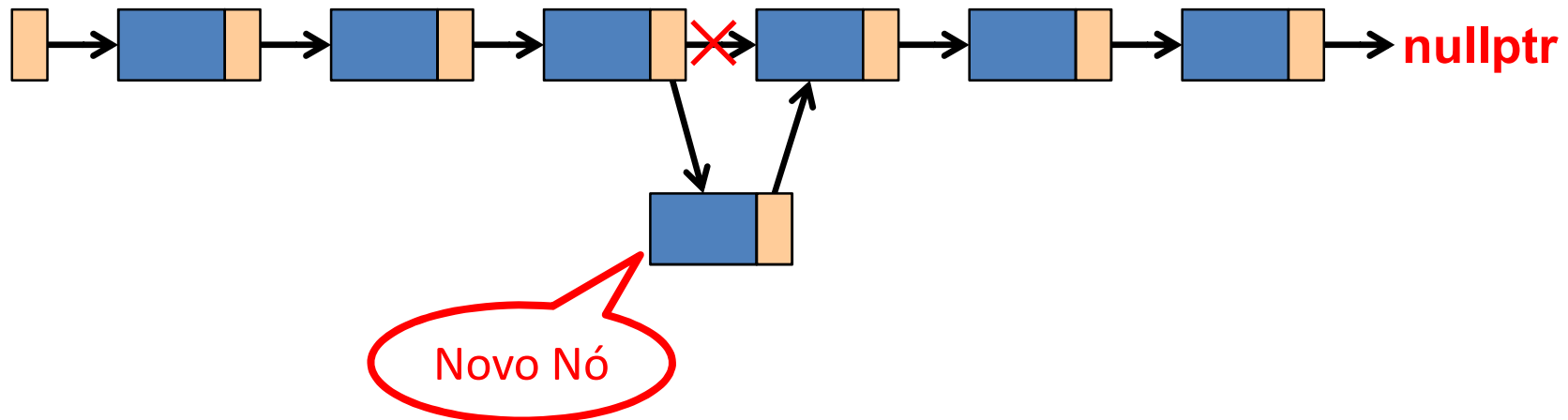
- Constituída por um conjunto de nós ligados entre si. Cada nó está ligado ao nó seguinte.





Lista Ligada

- Inserção (e remoção) de novos nós eficiente, só por manipulação de ponteiros, sem mover dados.





Nó duma Lista Ligada em C++

```
template <class T>

class CNoLista{
    public:
        T dados;
        CNoLista *proximo; //ponteiro para próximo nó
};
```



Definição de uma Lista Ligada

```
template <class T>

class CLista{
    CNoLista<T> *cabeca; // ponteiro para 1º elemento
public:
    CLista();
    ~CLista();

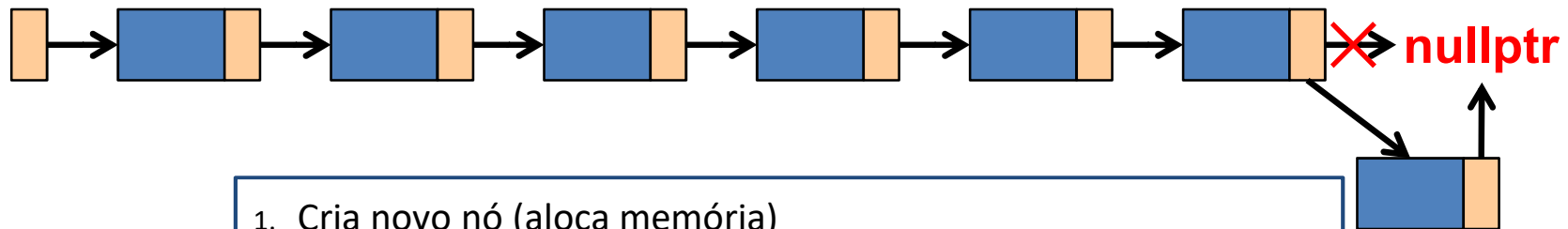
    void insereItem(const T &item);
    void apagaItem(const T &item);
    CNoLista<T> *procuraItem(const T &item);
    void mostraLista(void);

    ...
};
```



Inserção de um Nó na Lista

- Inserção do novo nó **no final** da lista:

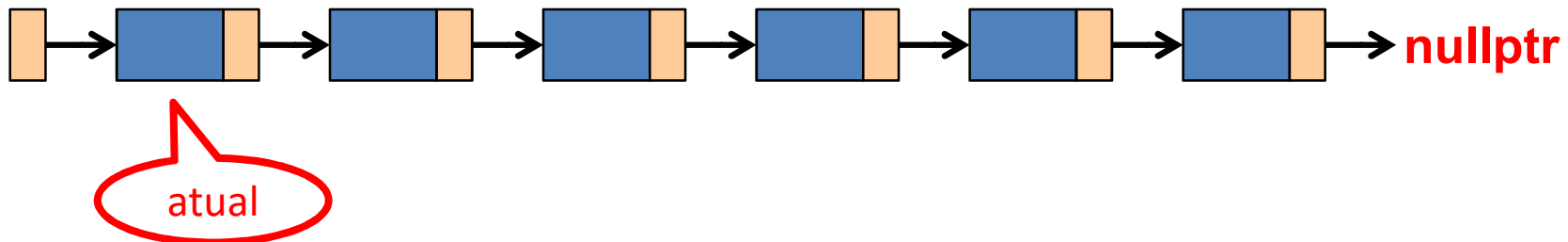


1. Cria novo nó (aloca memória)
2. Guarda os dados no novo nó
3. **Se** cabeça = nullptr **Então** cabeça aponta para o novo nó
4. **Senão:**
 5. Percorre a lista até chegar ao último nó
 6. Adiciona o novo nó no fim da lista
7. **Fim Se**



Inserção de um Nó na Lista

- Encontrar o último elemento:



```
atual = cabeca;  
while (atual->proximo != nullptr)  
    atual = atual->proximo;
```



Inserção de um Nó na Lista

```
template <class T> void CLista<T>::insereItem(const T &item) {
```

1

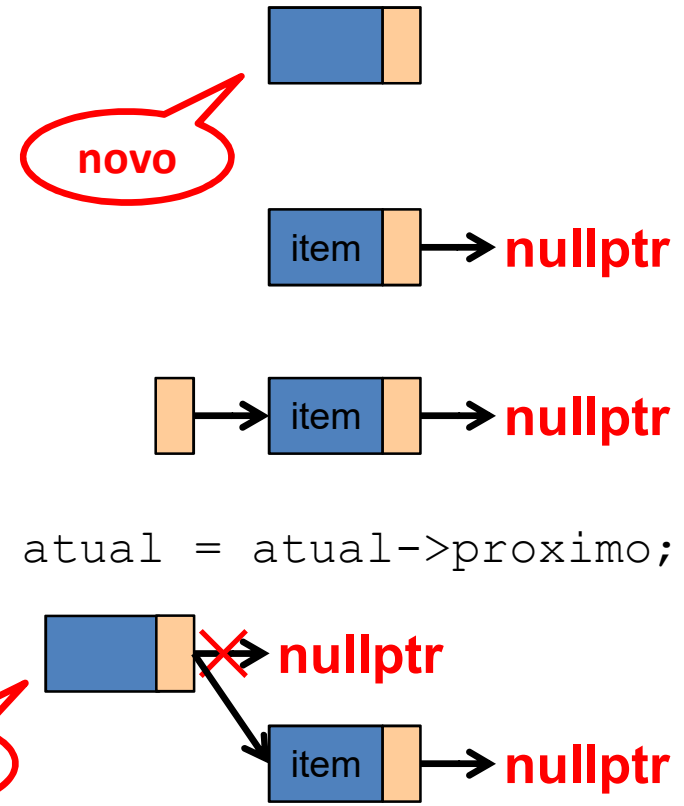
```
    CNoLista<T> *novo=new CNoLista<T>;  
    CNoLista<T> *atual;
```

2

```
    novo->dados = item;  
    novo->proximo = nullptr;
```

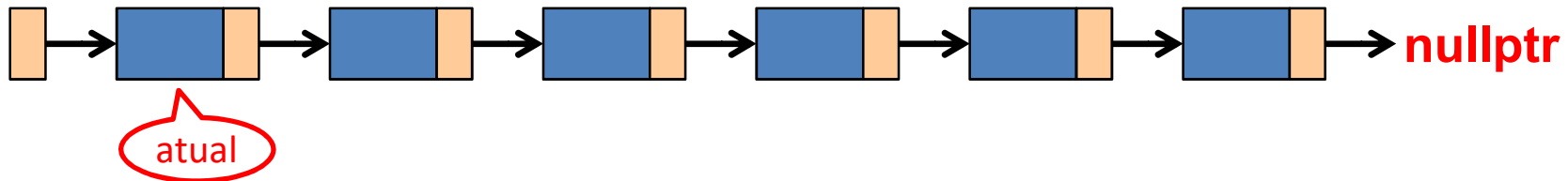
3

```
    if(cabeca == nullptr){  
        cabeca = novo;  
    } else {  
        atual = cabeca;  
        while(atual->proximo != nullptr) atual = atual->proximo;  
        atual->proximo = novo;  
    }  
}
```





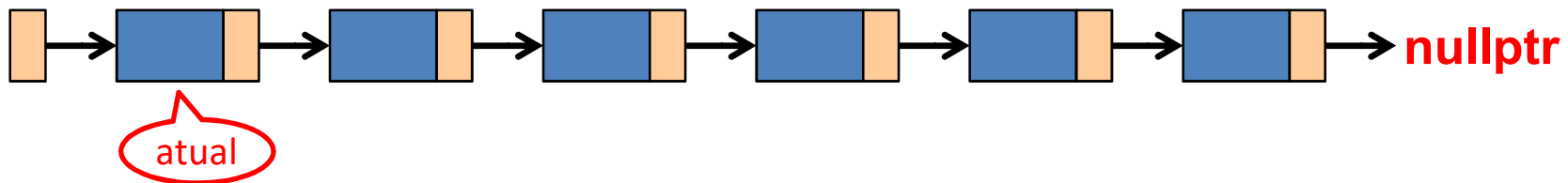
Mostrar a Lista Ligada (1)



```
template <class T>
void CLista<T>::mostraLista(void) const {
    if (cabeca == nullptr) {
        cout << "Lista Vazia..." << endl;
        return;
    }
    CNoLista<T> *atual = cabeca;
    while(atual != nullptr) {
        cout << atual->dados << ", ";
        atual = atual->proximo;
    }
    cout << "FIM" << endl;
}
```



Mostrar a Lista Ligada (2)



```
template <class T>
void CLista<T>::mostraLista(void) const {
    if (cabeca == nullptr) {
        cout << "Lista Vazia..." << endl;
        return;
    }
    for (CNoLista<T> *atual = cabeca;
        atual != nullptr;
        atual = atual->proximo) {
        cout << atual->dados << ", ";
    }
    cout << "FIM" << endl;
}
```

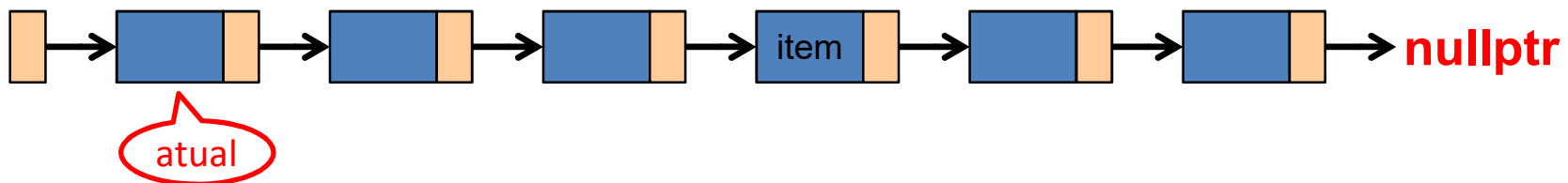


Estruturas Dinâmicas de Dados com Listas Ligadas

- Listas Ligadas (1ª e 2ª aulas) - continua
- Outros tipos de listas ligadas (2ª aula)
- Pilhas (3ª aula)
- Filas (3ª aula)
- Exemplos com Pilhas (3ª aula)
- Exemplos adicionais (final semestre)



Procurar um Item na Lista



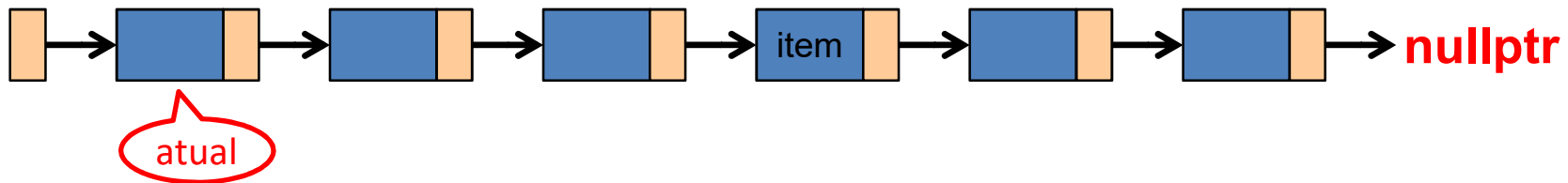
```

// retorna ptr para no' (nullptr se não existe)
template <class T>
CNoLista<T>* CLista<T>::procuraItem(const T &item) const{
    CNoLista<T> *atual = cabeca; // 1º no'
    while(atual != nullptr){ // percorre lista...
        if (atual->dados == item) return atual;
        atual = atual->proximo;
    }
    return(nullptr); // atual = nullptr
}

```



Procurar um Item na Lista (2)



```

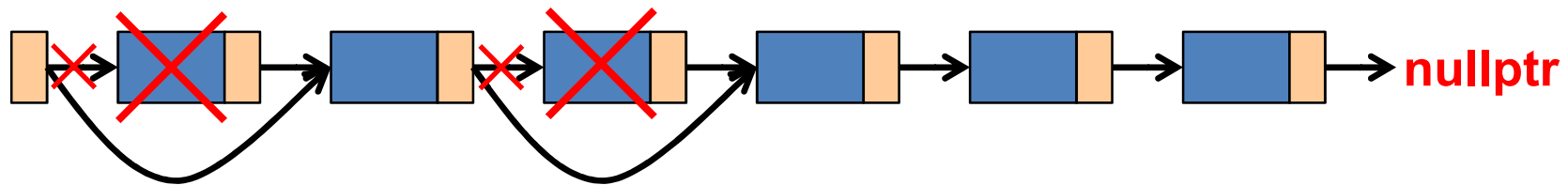
template <class T>
CNoLista<T>* CLista<T>::procuraItem(const T &item) const{

    CNoLista<T> *atual = cabeca;
    bool encontrado = false;

    while(atual != nullptr && !encontrado){
        if (atual->dados == item) encontrado = true;
        else atual = atual->proximo;
    }
    return(atual);
}
  
```



Remoção de um Nó da Lista



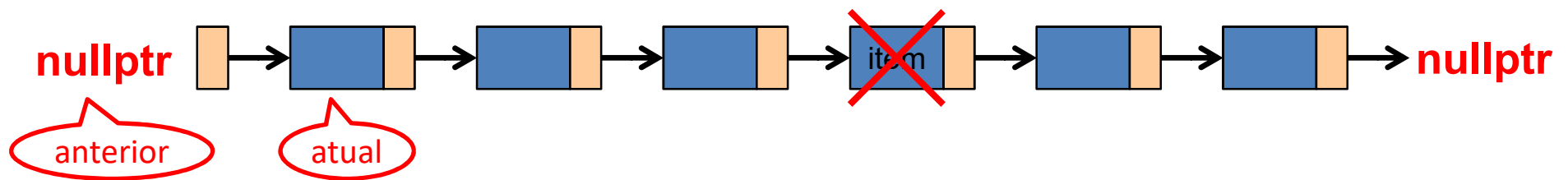
1. Procura na lista o nó que se pretende eliminar
2. **Se** o nó foi encontrado **Então**
3. **Se** o nó a eliminar é o 1º nó da lista
4. **Então** coloca **cabeca** a apontar para o nó a seguir
5. **Senão:**
6. Coloca o nó **anterior** ao nó a eliminar a apontar para o nó **a seguir** ao nó a eliminar
7. **Fim Se**
8. Liberta a memória ocupada pelo nó eliminado da lista
9. **Fim Se**

duas
situações
distintas a
considerar



Remoção de um Nó da Lista

- Procura na lista o nó que se pretende eliminar:



```

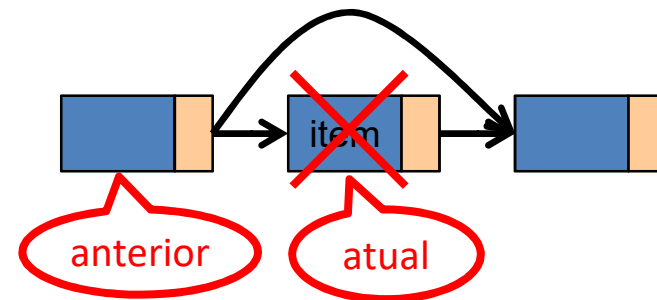
anterior = nullptr;
atual = cabeca;           // 1º nó
while (atual != nullptr){ // percorre lista...
    if(atual->dados == item){ //vai eliminar elemento aqui
    }
    anterior = atual;       // avança ptr anterior
    atual = atual->proximo;   // avança ptr atual
}

```



Remoção de um Nó da Lista

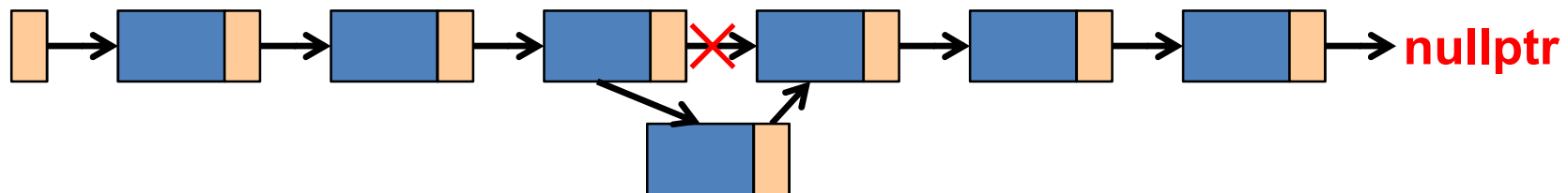
```
template <class T>
void CLista<T>::apagaItem(const T &item) {
    CNoLista<T> *anterior = nullptr, *atual = cabeca;
    while (atual != nullptr) {    // percorre lista...
        if (atual->dados == item) { // elimina elemento...
            if(anterior == nullptr) cabeca = atual->proximo;
            else anterior->proximo = atual->proximo;
            delete atual;    // liberta memória
            return;
        }
        anterior = atual;
        atual = atual->proximo;
    }
}
```





Inserção ordenada de nó na Lista

- Inserção do novo nó numa **lista ordenada**:

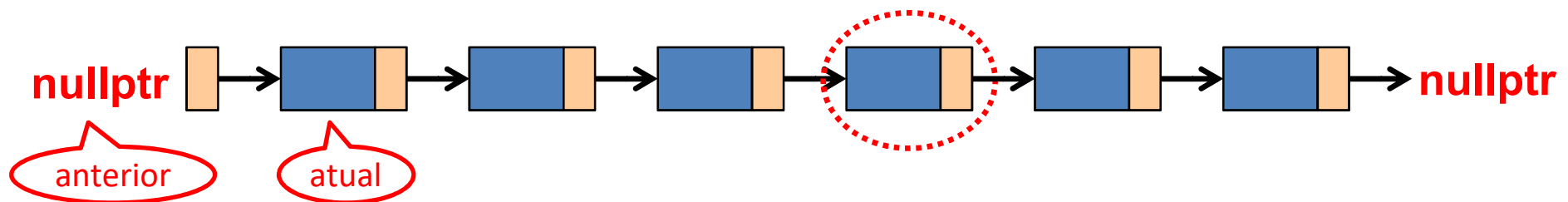


1. Cria novo nó (aloca memória)
2. Guarda os dados no novo nó
3. **Se** cabeça = nullptr **Então** cabeça aponta para o novo nó
4. **Senão**:
 5. Percorre a lista até encontrar a posição correta do novo nó
 6. Adiciona o novo nó na posição correta
7. **Fim Se**



Inserção ordenada de nó na Lista

- Encontrar a posição correta do novo elemento:



```
atual = cabeca;  
anterior = nullptr;  
while ((atual != nullptr) && (atual->dados < item)) {  
    anterior = atual;  
    atual = atual->proximo;  
}  
// vai inserir entre o anterior e o atual...
```



Inserção ordenada de nó na Lista

```
template <class T> void CLista<T>::insereItemOrdenado(const T &item) {
```

1

```
    CNoLista<T> *novo=new CNoLista<T>, *atual, *anterior;
```

2

```
    novo->dados = item;
    novo->proximo = nullptr;
```

```
    if (cabeca == nullptr) { cabeca=novo; return; }
```

```
    atual = cabeca;
```

```
    anterior = nullptr;
```

```
    while((atual!=nullptr) && (atual->dados<item)) {
```

```
        anterior = atual;
```

```
        atual = atual->proximo;
```

```
    }
```

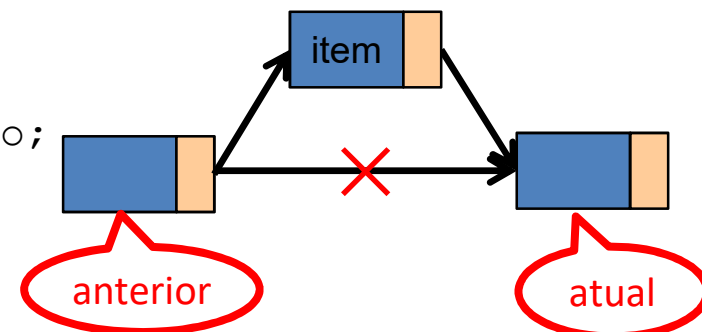
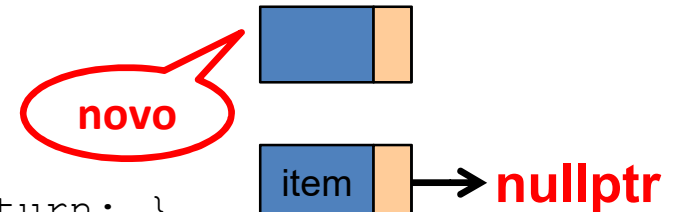
```
    if (anterior == nullptr) cabeca = novo;
```

```
    else anterior->proximo = novo;
```

```
    novo->proximo = atual;
```

3

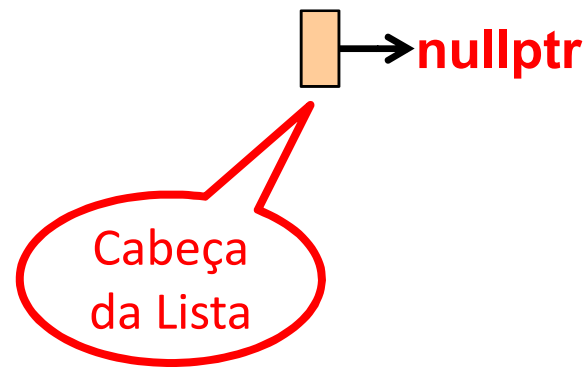
```
}
```





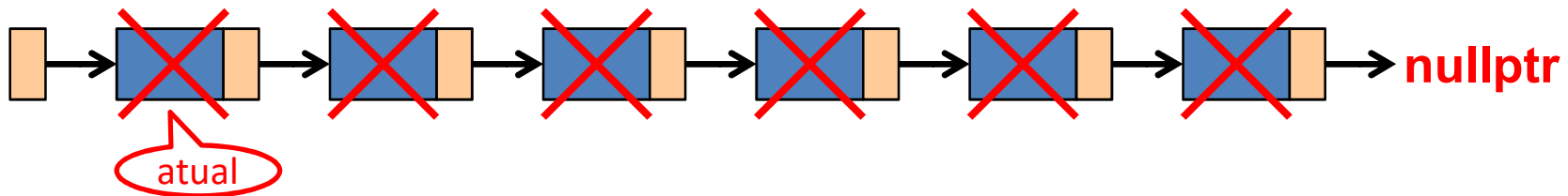
Construtor da Lista Ligada

```
template <class T> CLista<T>::CLista() {  
    cabeca = nullptr; // inicializa atributo  
}
```





Destruir da Lista Ligada



```
template <class T> CLista<T>::~~CLista() {
    CNoLista<T> *atual, *proximo;

    if (cabeca == nullptr) return; //redundante
    atual = cabeca;
    while (atual != nullptr){
        proximo = atual->proximo;
        delete atual;
        atual=proximo;
    }
    cabeca = nullptr;
}
```



Listas Ligadas

- Sobrecarga de Operadores:

- Operador de atribuição, e.g. `lista1 = lista2 = lista3;`

`CLista<T>& CLista<T>::operator = (const CLista<T> &lista)`

- Operador de concatenação, e.g. `lista1 += lista2;`

`CLista<T>& CLista<T>::operator += (const CLista<T> &lista)`



Sobrecarga do Operador =

```

template <class T>
CLista<T>& CLista<T>::operator = (const CLista<T> &lista){
    CNoLista<T> *atual, *proximo;
    if (cabeca != nullptr){
        atual = cabeca;
        while (atual != nullptr){
            proximo = atual->proximo;
            delete atual;
            atual = proximo;
        }
        cabeca = nullptr;
    }
    atual = lista.cabeca;
    while (atual!=nullptr){
        this->insereItem(atual->dados);
        atual = atual->proximo;
    }
    return (*this);
}

```

elimina
lista

lista2 = lista1;

atual

lista1

nullptr

lista2

nullptr

Inserir
novos
nós



Sobrecarga do Operador +=

```
template <class T>
CLista<T>& CLista<T>::operator += (const CLista<T> &lista) {
    // percorre a lista passada como parâmetro inserindo
    // cada elem no fim da própria lista (*this)
    CNoLista<T> *atual=lista.cabeca;

    while (atual != nullptr) {
        this->insereItem(atual->dados);
        atual=atual->proximo;
    }

    return (*this);
}
```



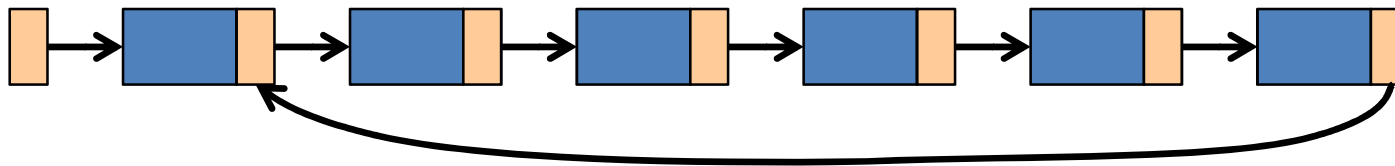
Estruturas Dinâmicas de Dados com Listas Ligadas

- Listas Ligadas (1ª e 2ª aulas)
- Outros tipos de listas ligadas (2ª aula)
- Exemplos adicionais (2ª aula)
- Pilhas (3ª aula)
- Filas (3ª aula)
- Exemplos com Pilhas (3ª aula)
- Exemplos adicionais (final semestre)

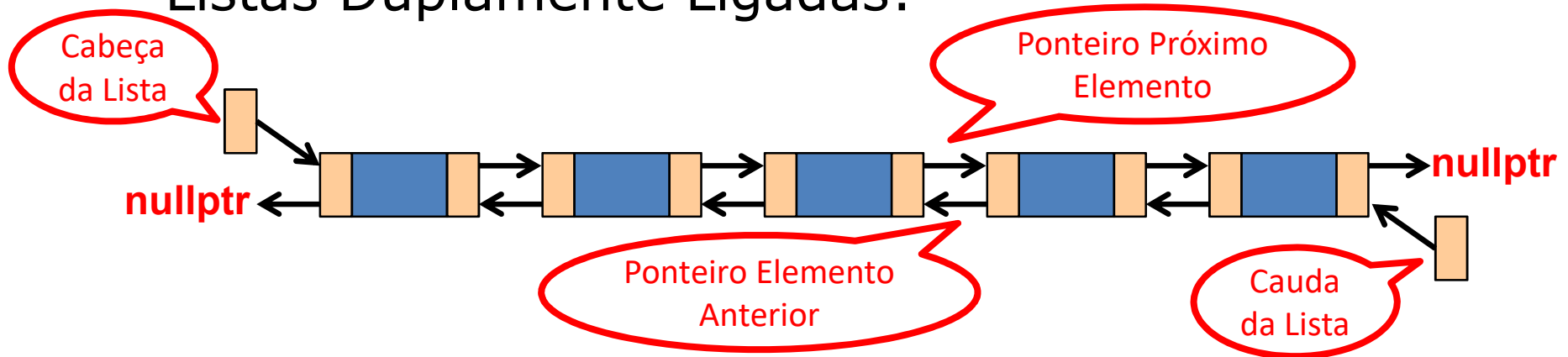


Outros Tipos de Listas Ligadas

■ Listas Circulares:

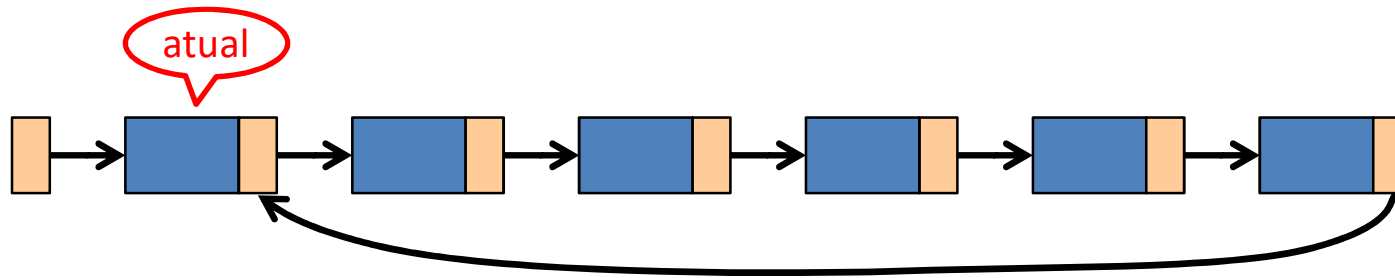


■ Listas Duplamente Ligadas:





Mostrar Lista Ligada Circular



```
template <class T>
void CListaCircular<T>::mostraLista(void) {
    CNoLista<T> *atual = cabeca;

    if (cabeca == nullptr) cout << "Lista Vazia..." << endl;
    else {
        do{
            cout << atual->dados << ", ";
            atual = atual->proximo;
        } while(atual != cabeca);    // enquanto != 1º
        cout << "FIM" << endl;
    }
}
```




Nó de uma Lista Duplamente Ligada em C++

```
template <class T>

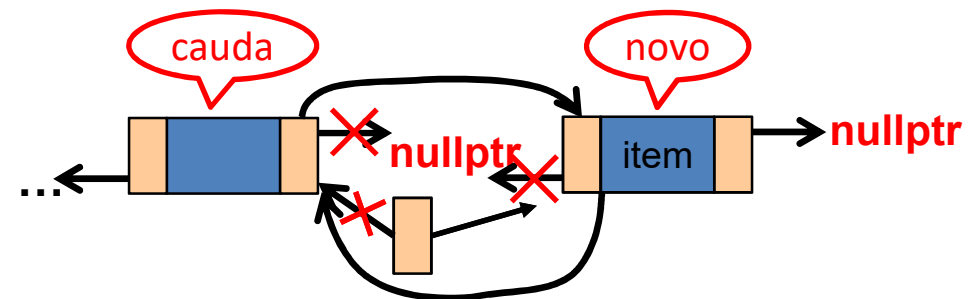
class CNoLista2{
    public:
        T dados;
        CNoLista2 *proximo; //ponteiro para próximo nó
        CNoLista2 *anterior; //ponteiro para nó anterior
};
```



Inserção de um Nó na Cauda da Lista Duplamente Ligada

```
template <class T>
void CLista2<T>::insereItem(const T &item) {
    1 CNoLista2<T> *novo=new CNoLista2<T>;
      novo->dados=item;
      novo->proximo=nullptr;
      novo->anterior=nullptr;

      if(cabeca == nullptr){ //lista vazia
          cabeca=novo; cauda=novo;
      }
      2 else { //lista não vazia
        cauda->proximo=novo;
        novo->anterior=cauda;
        cauda=novo;
      }
    }
```





Estruturas de Dados Variáveis

Solução: **Base de Dados Relacional**

Clientes		
ClienteID	Nome	Telefone
123	Rachel Ingram	555-861-2025
456	James Wright	555-403-1659 555-776-4100
789	Maria Fernandez	555-808-9633



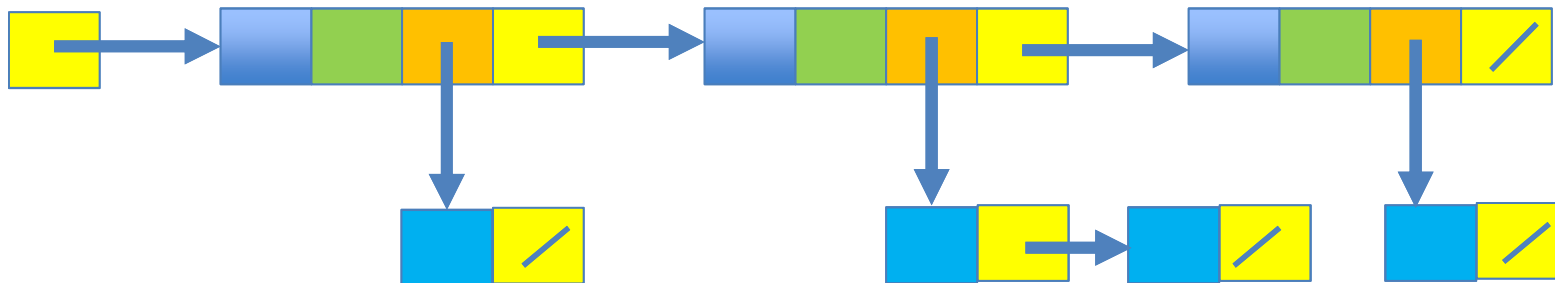
Clientes	
ClienteID	Nome
123	Rachel
456	James
789	Maria

Telefones	
ClienteID	Telefone
123	555-861-2025
456	555-403-1659
456	555-776-4100
789	555-808-9633



Estruturas Dinâmicas de Dados Variáveis com Listas Ligadas

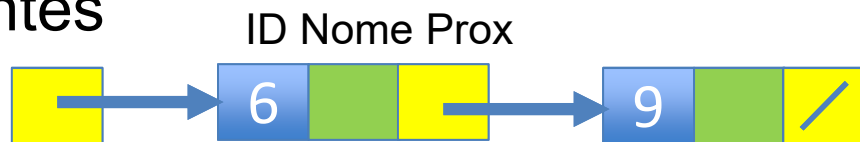
Clientes		
ClienteID	Nomes	Telefones
123	Rachel Ingram	555-861-2025
456	James Wright	555-403-1659 555-776-4100
789	Maria Fernandez	555-808-9633





Estruturas Dinâmicas de Dados Variáveis com Listas Ligadas

Clientes



Clientes	
ClienteID	Nome
123	Rachel
456	James
789	Maria

Telefones



Telefones	
ClienteID	Telefone
123	555-861-2025
456	555-403-1659
456	555-776-4100
789	555-808-9633



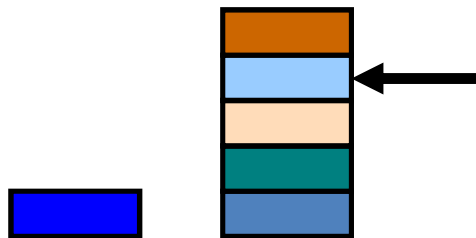
Estruturas Dinâmicas de Dados com Listas Ligadas

- Listas Ligadas (1ª e 2ª aulas)
- Outros tipos de listas ligadas (2ª aula)
- Pilhas (3ª aula)
- Filas (3ª aula)
- Exemplos com Pilhas (3ª aula)
- Exemplos adicionais (final semestre)



Pilha

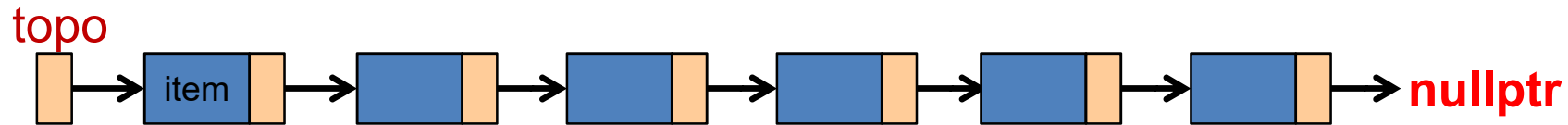
- O funcionamento de uma pilha (*stack*) é análogo a uma pilha de livros:
 - **Adicionamos** um livro colocando-o **no topo**.
 - Se retirarmos um livro, **retiramos o livro no topo**, i.e. o **último** que foi colocado – *last in, first out* ou **LIFO**.



- **Duas operações elementares:**
 - **Push** – insere um elemento no **topo** da pilha;
 - **Pop** – retira o elemento no **topo** da pilha.



Definição de uma Pilha em C++



```
template <class T>
class CNoPilha{
public:
    T dados;
    CNoPilha *proximo;
};
```

```
template <class T>
class CPilha{
    CNoPilha<T> *topo; // ponteiro para topo
public:
    CPilha(void);
    ~CPilha(void);

    void push(const T &item);
    bool pop(T &item);
    void mostraPilha(void);
    bool pilhaVazia(){return (topo==nullptr); }
    ...
};
```

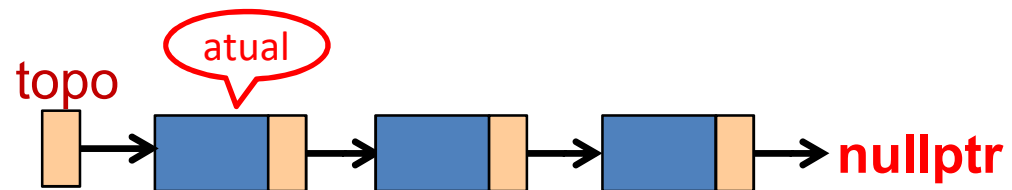



Construtor/Destrutor de CPilha

- **Construtor:**

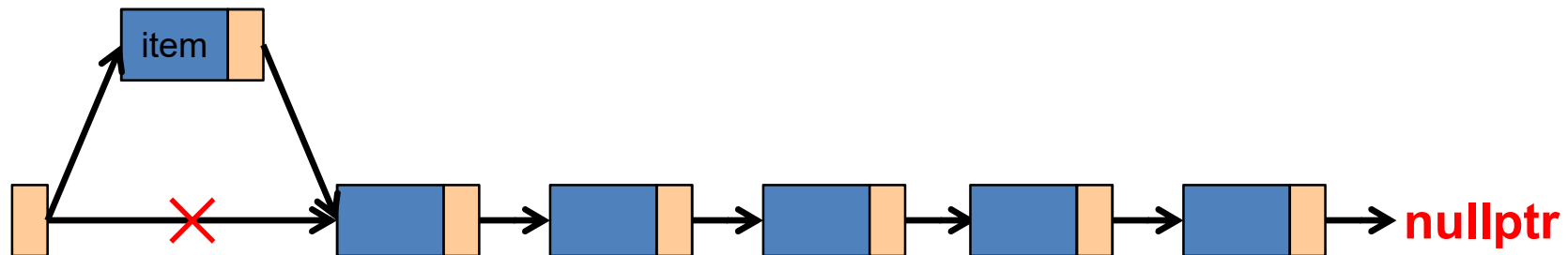
```
template <class T> CPilha<T>::CPilha()  
{ topo = nullptr; }
```
- **Destrutor:**

```
template <class T> CPilha<T>::~~CPilha() {  
    CNoPilha<T> *atual;  
  
    while (topo!=nullptr) {  
        atual = topo;  
        topo = topo->proximo;  
        delete atual;  
    }  
}
```





Pilha – Método Push

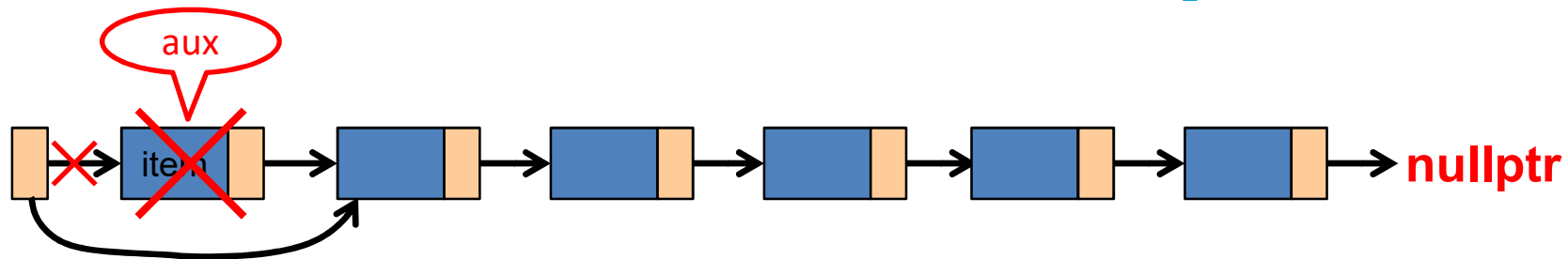


```
template <class T>
void CPilha<T>::push(const T &item) {
    CNoPilha<T> *novo=new CNoPilha<T>;

    novo->dados = item;
    novo->proximo=topo;
    topo=novo;
}
```



Pilha – Método Pop



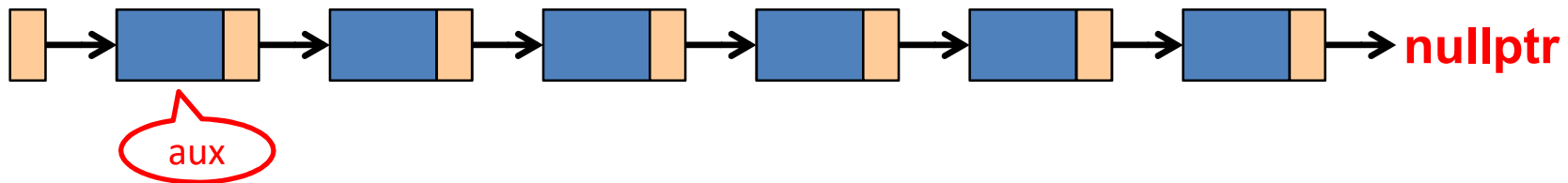
```
template <class T>
bool CPilha<T>::pop(T &item) {
    CNoPilha<T> *aux;

    if(topo == nullptr)    return(false);    // pilha vazia

    item = topo->dados;
    aux = topo;
    topo = topo->proximo;
    delete(aux);
    return(true);
}
```



Pilha – Mostrar Conteúdo



```
template <class T>
void CPilha<T>::mostraPilha(void) {
    if(topo==nullptr) cout << "Pilha Vazia..." << endl;
    for(CNoPilha<T> *aux=topo; aux!=nullptr; aux=aux->proximo)
        cout << aux->dados << endl;
}
```

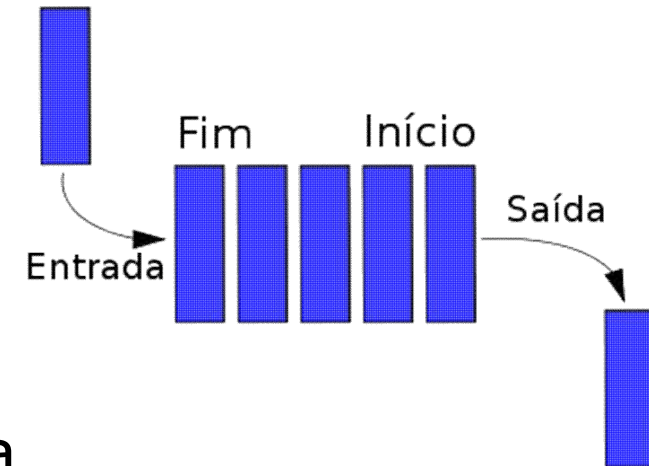


Estruturas Dinâmicas de Dados com Listas Ligadas

- Listas Ligadas (1ª e 2ª aulas)
- Outros tipos de listas ligadas (2ª aula)
- Exemplos com listas (2ª aula)
- Pilhas (3ª aula)
- **Filas (3ª aula)**
- Exemplos com Pilhas (3ª aula)
- Exemplos adicionais listas e filas (final semestre)



Fila (Queue)

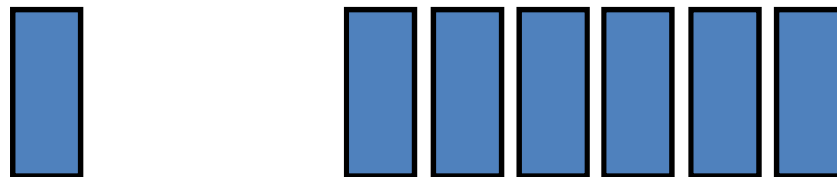


- Outra estrutura de dados ligada muito utilizada.
- Principal diferença entre Pilha e Fila
é a ordem de saída dos elementos: na fila, o “primeiro a entrar é o primeiro a sair” – *first in first out* ou **FIFO**.
- A *inserção* de um novo elemento é sempre feita no *final* da fila e o primeiro elemento que se pode *remover* é o que está no *início* da fila.



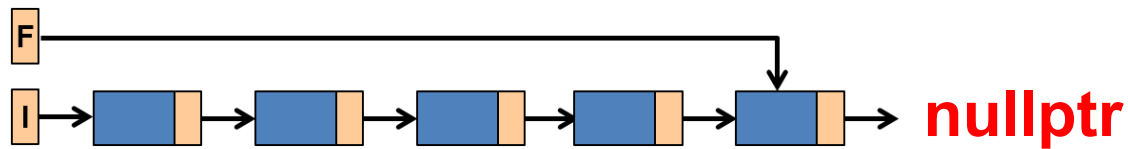
Fila

- Analogia – Fila de Espera
 - A estrutura de uma fila é análoga ao conceito de fila de espera que usamos no nosso dia a dia:
 - Quem está no início da fila é o primeiro a ser atendido, ou seja o primeiro a sair da fila.
 - Quem chega, fica no final da fila.





Definição de uma Fila em C++



```
template <class T>
class CNoFila{
public:
    T dados;
    CNoFila *proximo;
};
```

```
template <class T>
class CFila{
    CNoFila<T> *inicio; // ponteiro p/ início
    CNoFila<T> *fim;    // ponteiro p/ fim
public:
    CFila(void);
    ~CFila(void);

    void insere(const T &item);
    bool retira(T &item);
    void mostraFila(void);
    bool filaVazia(){return(inicio==nullptr);}
    ...
};
```




Construtor/Destrutor de CFila

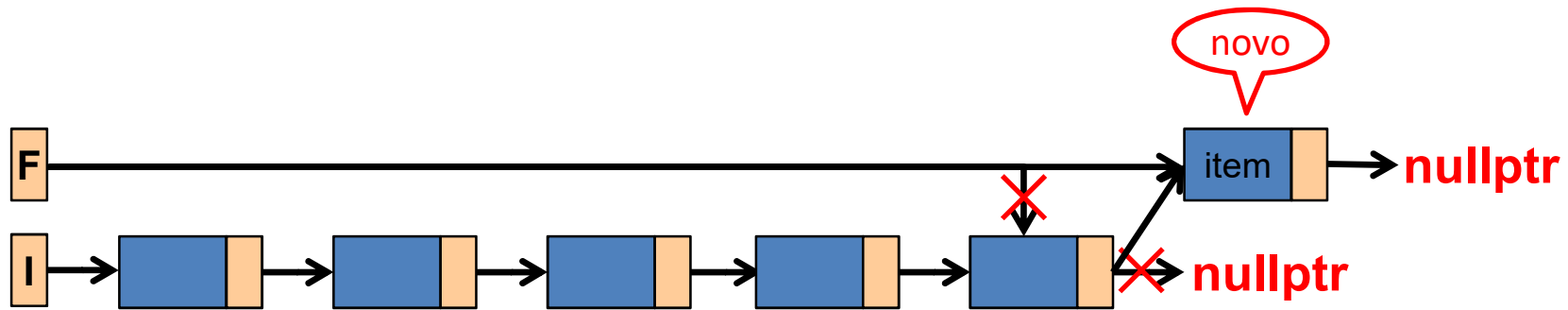
- **Construtor:**

```
template <class T> CFila<T>::CFila()  
{ inicio = nullptr; fim = nullptr; }
```
- **Destrutor:**

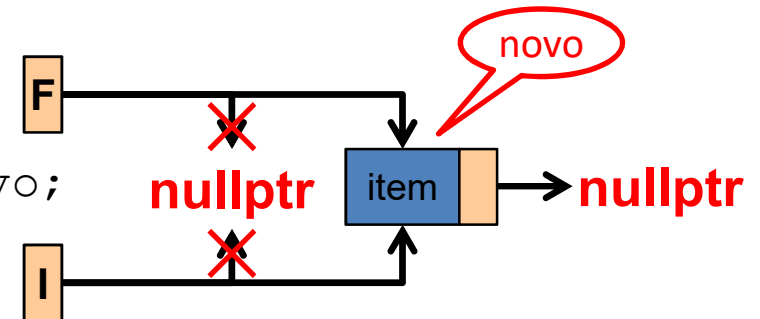
```
template <class T> CFila<T>::~~CFila() {  
    CNoFila<T> *atual, *proximo;  
  
    if(inicio == nullptr) return;  
    atual = inicio;  
    while (atual!=nullptr) {  
        proximo = atual->proximo;  
        delete atual;  
        atual = proximo;  
    }  
    inicio = nullptr;  
    fim = nullptr;  
}
```



Fila – Inserção de um Elemento

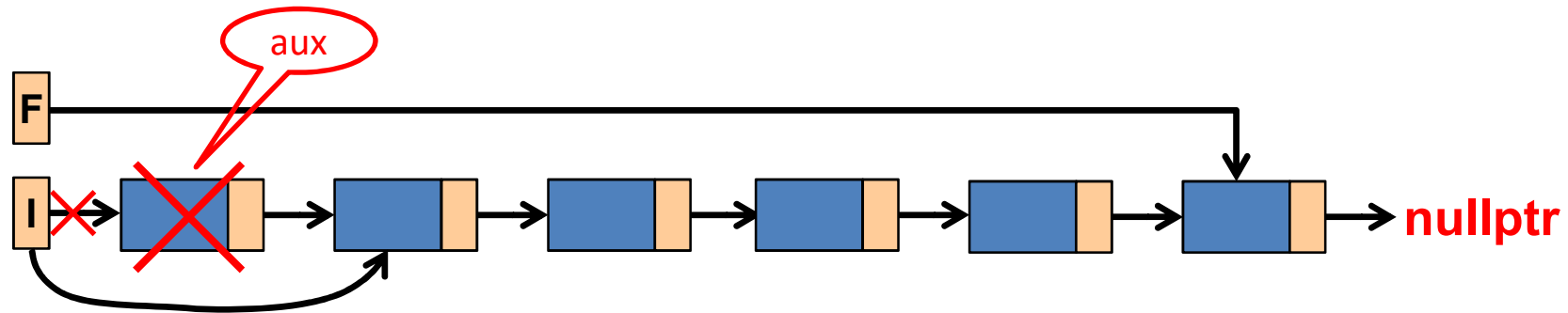


```
template <class T> void CFila<T>::insere(const T &item) {
    CNoFila<T> *novo = new CNoFila<T>;
    novo->dados = item;
    novo->proximo = nullptr;
    if (inicio == nullptr) inicio = novo;
    else fim->proximo = novo;
    fim = novo;
}
```



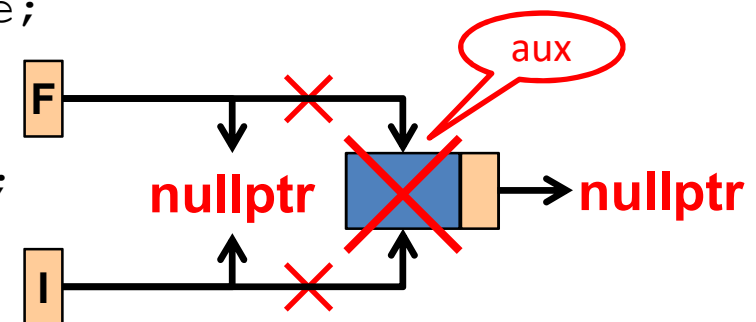


Fila – Remoção de um Elemento



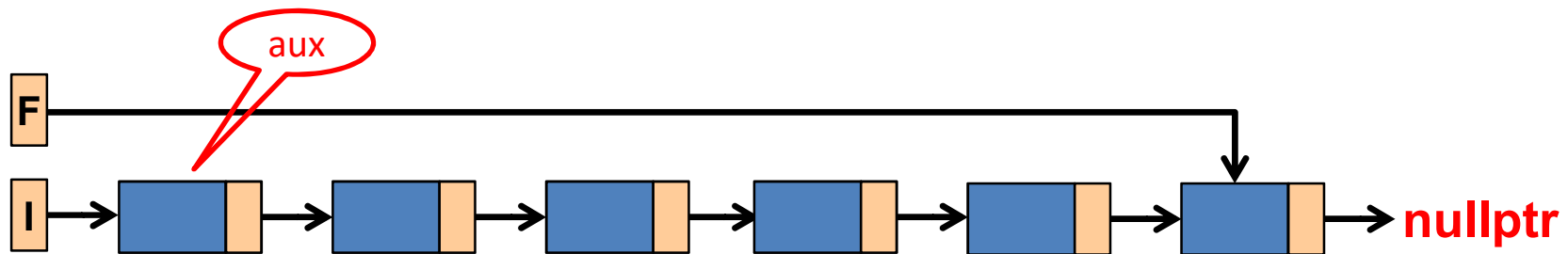
```
template <class T> bool CFila<T>::retira (T &item) {
    CNoFila<T> *aux = inicio;

    if (inicio == nullptr) return false;
    item = inicio->dados;
    inicio = aux->proximo;
    if (inicio == nullptr) fim=nullptr;
    delete aux;
    return true;
}
```





Fila – Mostrar Conteúdo



```

template <class T> void CFila<T>::mostraFila(void) const{
    CNoFila<T> *aux;

    if(inicio == nullptr) cout << "A Fila está vazia." << endl;
    else{
        aux = inicio;
        while(aux != nullptr){
            cout << aux->dados << endl;
            aux = aux->proximo;
        }
    }
}

```



Exemplos com Pilhas

1. Detetar capicuas
2. Avaliar coerência nos parêntesis de uma expressão matemática
3. Calcular expressões matemáticas



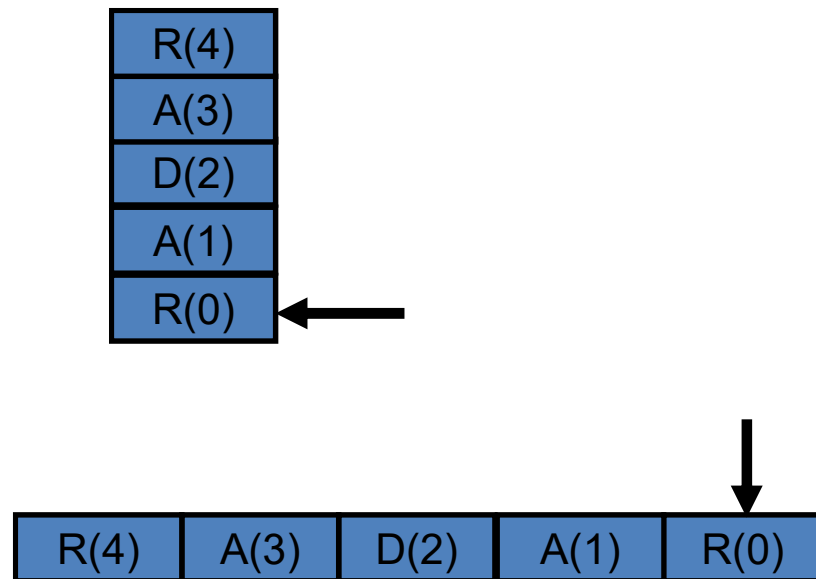
Ex. 1 – Detetar Capicuas

- Determinar se uma cadeia de caracteres é uma capicua.
 - Exemplos:
 - RADAR
 - ANA
 - 2002
- Algoritmo:
 1. Criar uma pilha e uma fila com os caracteres da string.
 2. Ao retirar os caracteres da pilha e da fila, aqueles vêm na ordem inversa no 1º caso e na ordem normal no segundo.
 3. Se a sequência obtida for igual, é detetada uma capicua.



Ex. 1 – Detetar Capicuas

- Exemplo:
"RADAR"



```
bool verificaSeCapicua(char *s) {
    int i;
    char c1, c2;
    CPilha<char> pilha;
    CFila<char> fila;

    for(i=0; i<strlen(s); i++) {
        pilha.push(s[i]);
        fila.insere(s[i]);
    }
    for(i=0; i<strlen(s); i++) {
        pilha.pop(c1);
        fila.retira(c2);
        if(c1 != c2) return(false);
    }
    return(true);
}
```



Ex. 2 – Verificar Parêntesis

- São habitualmente usadas pilhas para avaliar expressões matemáticas.
- Exemplo simples: verificar coerência nos parêntesis de uma expressão matemática.
 - Ex.: $((A + B * (C + 2)) * (D + E))$
 - **Algoritmo:** utilizar uma pilha para verificar se o número de parêntesis é o correto.
 1. Inserir um item na pilha por cada '(' e remover um item da pilha por cada ')'.
 2. Se a pilha não estiver vazia sempre que apareça ')' e se no final a pilha estiver vazia, o número de parêntesis é o correto.

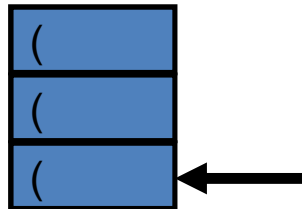


Ex. 2 – Verificar Parêntesis

■ Exemplo:



`((A + B * (C + 2)) * (D + E))`



```
bool verificaParentesis(char *s) {  
    CPilha<char> pilha;  
    unsigned int i;  
    char c;  
  
    for(i=0; i<strlen(s); i++){  
        if(s[i] == '(' )  
            pilha.push('(');  
        if(s[i] == ')' ){  
            if(!pilha.pop(c))  
                return false; //falta '('  
        }  
    }  
    if(pilha.pilhaVazia())  
        return true;  
    else return false; //falta ')'   
}
```



Ex. 3 – Calcular Expressões Matemáticas

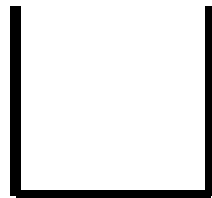
- Um exemplo um pouco mais complexo, e.g. calcular:
 - Ex.: $(((4 + 5) * (2 + 3)) / (1 + 4))$
- **Algoritmo:** assumindo que cada operação é delimitada por parêntesis, utiliza-se uma **pilha para operandos** e uma **pilha para operações**.
 1. Sempre que se lê um **novo número** coloca-se na **pilha de operandos**.
 2. Sempre que surge uma **nova operação** coloca-se na **pilha das operações**.
 3. Quando surge `)', **remove-se** da pilha de operações a **operação** e os **dois operandos** que estão no topo da pilha de operandos.
 4. Realiza-se a operação e insere-se o **resultado na pilha de operandos**.
 5. Todos os outros caracteres são ignorados.



Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



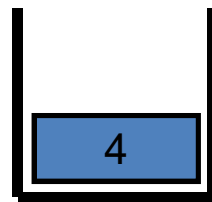
pilhaOper



Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



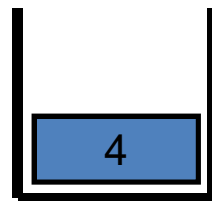
pilhaOper



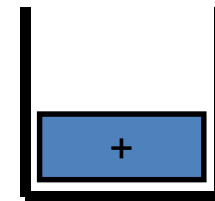
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



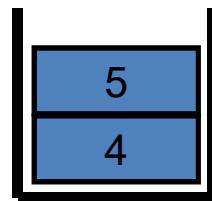
pilhaOper



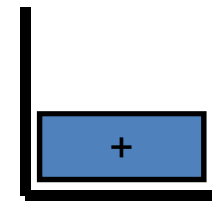
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



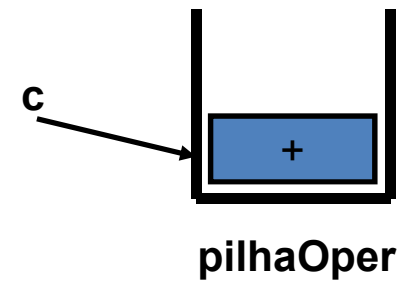
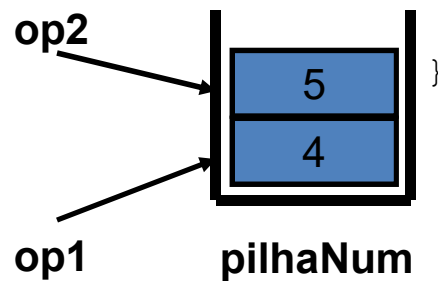
pilhaOper



Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

```
if(c == ')') {  
    pilhaNum.pop(op2);  
    pilhaNum.pop(op1);  
    pilhaOper.pop(c);  
    switch(c) {  
        case '+': pilhaNum.push(op1+op2); break;  
        case '-': pilhaNum.push(op1-op2); break;  
        case '*': pilhaNum.push(op1*op2); break;  
        case '/': pilhaNum.push(op1/op2); break;  
    }  
}
```

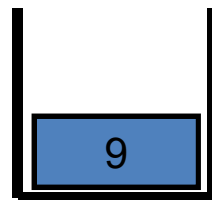




Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



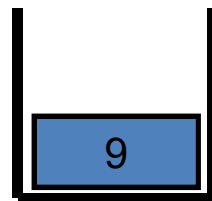
pilhaOper



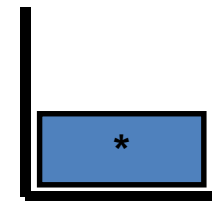
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



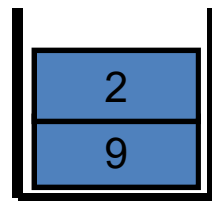
pilhaOper



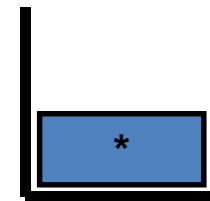
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



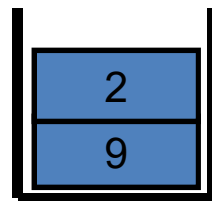
pilhaOper



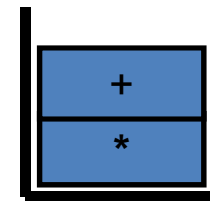
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



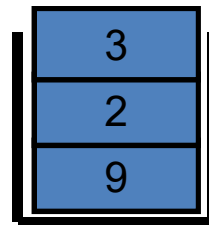
pilhaOper



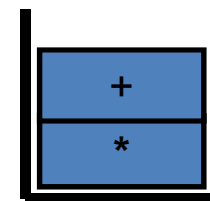
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



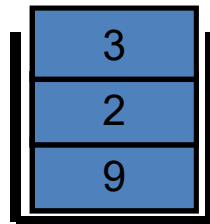
pilhaOper



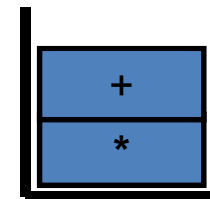
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



PilhaNum



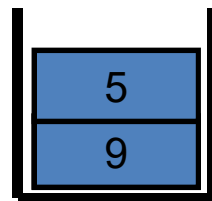
PilhaOper



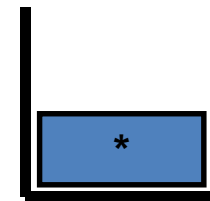
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



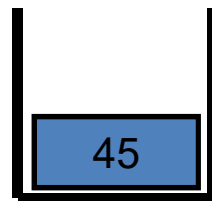
pilhaOper



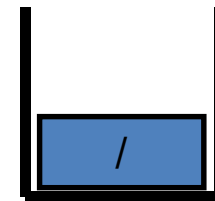
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum

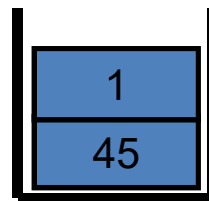


pilhaOper

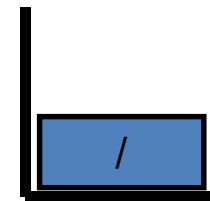


Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$



pilhaNum



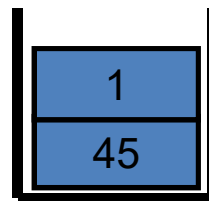
pilhaOper



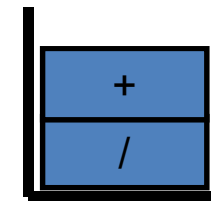
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum

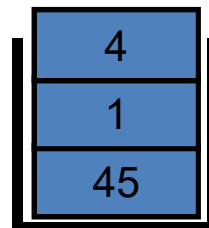


pilhaOper

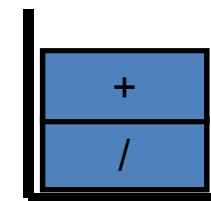


Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$



pilhaNum



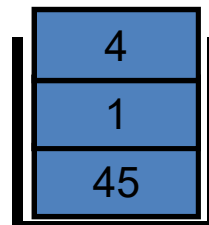
pilhaOper



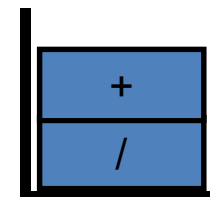
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



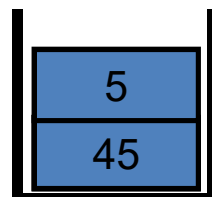
pilhaOper



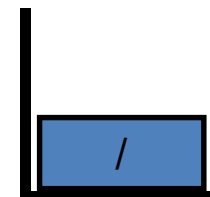
Ex. 3 – Calcular Expressões Matemáticas

Exemplo: $(((4 + 5) * (2 + 3)) / (1 + 4))$

↑



pilhaNum



pilhaOper



Ex. 3 – Calcular Expressões Matemáticas

```
double calcExpressao(char *s){
    unsigned int i;
    char c;
    double num, op1, op2;
    CPilha<char> pilhaOper;
    CPilha<double> pilhaNum;
    bool eNumero=false;

    for(i=0; i < strlen(s); i++){
        c = s[i];
        if (isdigit(c)){
            if(!eNumero){
                eNumero = true;
                num = 0;
            }
            num=num*10+(c-'0');
        }else{ // c is not digit
            if(eNumero){
                pilhaNum.push(num);
                eNumero = false;
            }

```

((1234+...

```
        else if (strchr("+-*/", c) != nullptr)
            pilhaOper.push(c);
        else{
            if(c == '('){
                pilhaNum.pop(op2);
                pilhaNum.pop(op1);
                pilhaOper.pop(c);
                switch(c){
                    case '+': pilhaNum.push(op1+op2); break;
                    case '-': pilhaNum.push(op1-op2); break;
                    case '*': pilhaNum.push(op1*op2); break;
                    case '/': pilhaNum.push(op1/op2); break;
                }
            }
        } // fim else "c is not digit"
    } // fim for
    pilhaNum.pop(num);
    return(num);
}
```



Estruturas Dinâmicas de Dados com Listas Ligadas

- Listas Ligadas (1ª e 2ª aulas)
- Outros tipos de listas ligadas (2ª aula)
- Exemplos com listas (2ª aula)
- Pilhas (3ª aula)
- Filas (3ª aula)
- Exemplos com Pilhas (3ª aula)
- Exemplos adicionais listas e filas (final semestre)



6. Árvores Binárias

A partir da próxima aula...



Estruturas Dinâmicas de Dados com Listas Ligadas

- Listas Ligadas (1ª e 2ª aulas)
- Outros tipos de listas ligadas (2ª aula)
- Exemplos com listas (2ª aula)
- Pilhas (3ª aula)
- Filas (3ª aula)
- Exemplos com Pilhas (3ª aula)
- Exemplos adicionais listas e filas (final semestre)



Estruturas de Dados com Listas Ligadas

Exemplos Adicionais

4. Comparar listas ligadas
5. Intercalar listas ligadas
6. Elementos comuns de duas listas ligadas
7. Eliminar penúltimo elemento de uma fila
8. Prioridade a pares numa fila de números



Ex. 4 – Comparar Listas Ligadas

- Definição de uma lista ligada de números inteiros:

```
class CNoLista{  
    public:  
        int dados;  
        CNoLista *proximo;  
};  
  
class CListaInteiros{  
    CNoLista *cabeca;  
    public:  
        CListaInteiros(void);  
        ~CListaInteiros(void);  
  
        void insereItem(const int);  
        void apagaItem(const int);  
};
```

- Implemente a sobrecarga do **operador >** para que este devolva o resultado da comparação entre o objeto original (1ª lista) e outro objeto da mesma classe (2ª lista), definida através da **comparação do maior número armazenado** em cada uma das listas.



Ex. 4 – Comparar Listas Ligadas

- Caso de teste:

- $L1 = 10 \rightarrow 20 \rightarrow 45 \rightarrow 36 \rightarrow 12 \rightarrow \text{nullptr}$
- $L2 = 17 \rightarrow 35 \rightarrow 24 \rightarrow 16 \rightarrow \text{nullptr}$
- $L1 > L2$ é uma proposição verdadeira porque o maior elemento de L1 (45) é maior do que o maior elemento de L2 (35).

- Passos da solução:

1. Implementação de um método adicional para devolver o maior valor da lista.
2. Implementação da sobrecarga do operador $>$.



Ex. 4 – Comparar Listas Ligadas

- Método para devolver o maior valor da lista

```
int CListaInteiros::maiorValor(void) {  
    if (cabeca == nullptr) return(INT_MIN); // menor int  
    int maior = cabeca->dados;  
    CNoLista *aux = cabeca->proximo;  
    while (aux != nullptr) {  
        if (aux->dados > maior) maior = aux->dados;  
        aux = aux->proximo;  
    }  
    return(maior);  
}
```



Ex. 4 – Comparar Listas Ligadas

- Sobrecarga do operador >

```
bool CListaInteiros::operator > (const
                                CListaInteiros &l) {
    // se nenhuma das listas está vazia...
    if ( (cabeca != nullptr) && (l.cabeca != nullptr) )
        return ( maiorValor() > l.MaiorValor() );

    // se só 2ª lista vazia...
    if ( (cabeca != nullptr) && (l.cabeca == nullptr) )
        return true;

    // ambas as listas vazias ou só 1ª lista vazia...
    return false;
}
```



Ex. 4 – Comparar Listas Ligadas

- Ou, ainda mais simples:

```
bool CListaInteiros::operator > (CListaInteiros &l) {  
    return ( maiorValor() > l.MaiorValor() );  
}
```

- Funciona porque o método `maiorValor()` testa se a lista está vazia e nesse caso retorna o menor número da gama `int (INT_MIN)`. Parte-se do princípio de que nenhuma lista tem o inteiro `INT_MIN`.



Ex. 5 – Intercalar Listas Ligadas

- Definição de uma lista ligada de caracteres:

```
class CNoLista{  
    public:  
        char dados;  
        CNoLista *proximo;  
};
```

```
class CListaCaracteres{  
    CNoLista *cabeca;  
    public:  
        CListaCaracteres(void);  
        ~CListaCaracteres(void);  
  
        void insereItem(char c);  
        void apagaItem(char c);  
};
```

- Implemente a sobrecarga do **operador +=** para que este permita **intercalar** os caracteres contidos na **lista original** com os caracteres da **segunda lista**.



Ex. 5 – Intercalar Listas Ligadas

- Caso de teste:

- $L1 = a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow \text{nullptr}$
- $L2 = u \rightarrow v \rightarrow w \rightarrow x \rightarrow y \rightarrow z \rightarrow \text{nullptr}$
- $L1 \oplus L2$ deve ter como resultado:

$L1 = a \rightarrow u \rightarrow b \rightarrow v \rightarrow c \rightarrow w \rightarrow d \rightarrow x \rightarrow e \rightarrow y \rightarrow z \rightarrow \text{nullptr}$

- A 2ª lista não deve ser modificada.
- Se uma lista tiver mais elementos do que a outra, os elementos que não puderem ser intercalados devem ser posicionados no final da lista.



Ex. 5 – Intercalar Listas Ligadas

■ Sobrecarga do operador +=

```
void CListaCaracteres::  
    operator += (CListaCaracteres &l) {  
  
    CNoLista *aux1, *aux2, *novo;  
    aux1 = cabeca;  
    aux2 = l.cabeca;  
    while ( (aux1!=nullptr) &&  
            (aux2!=nullptr) ) {  
        novo=new CNoLista;  
        novo->dados = aux2->dados ;  
        novo->proximo = aux1->proximo ;  
        aux1->proximo = novo;  
        aux1 = novo->proximo;  
        aux2 = aux2->proximo;  
    }  
    // continua ao lado →
```

```
//Ciclo anterior termina quando  
// apenas uma lista ainda tem  
// elementos.  
//Se for a 1ª não é preciso  
// fazer nada.  
//Se for a 2ª, insere no fim da  
// lista...
```

```
while (aux2 != nullptr) {  
    insereItem(aux2->dados);  
    aux2 = aux2->proximo;  
}  
}
```



Ex. 6 – Elementos comuns de duas listas ligadas

- Definição de uma lista ligada de números inteiros:

```
class CNoLista{  
    public:  
        int numero;  
        CNoLista *proximo;  
};
```

```
class CListaInteiros{  
    CNoLista *cabeca;  
    public:  
        CListaInteiros(void);  
        ~CListaInteiros(void);  
  
        void insereItem(const int item);  
        void apagaItem(const int item);  
        CListaInteiros operator=  
            (CListaInteiros l){  
  
        };
```

- Implemente a sobrecarga do operador – que permita obter uma nova lista composta apenas pelos elementos comuns da lista original e da 2ª lista, sendo que esta não deve ser alterada.



Ex. 6 – Elementos comuns de duas listas ligadas

- Caso de teste:
 - $L1 = 25 \rightarrow 46 \rightarrow 13 \rightarrow 26 \rightarrow 7 \rightarrow \text{nullptr}$
 - $L2 = 16 \rightarrow 7 \rightarrow 46 \rightarrow 18 \rightarrow 25 \rightarrow 12 \rightarrow 39 \rightarrow 13 \rightarrow \text{nullptr}$
 - $L3 = L1 - L2 = 25 \rightarrow 46 \rightarrow 13 \rightarrow 7 \rightarrow \text{nullptr}$
- Passos da solução:
 1. Implementação de um método adicional para verificar se um determinado número existe ou não na lista.
 2. Implementação da sobrecarga do operador $-$.



Ex. 6 – Elementos comuns de duas listas ligadas

- Método para verificar se um número existe na lista

```
bool CListaInteiros::procuraItem(const int item){  
    CNoLista *atual = cabeca;  
  
    while (atual != nullptr) { // percorre lista  
        if (atual->numero == item)  
            return true; // encontrou, devolve true  
        atual = atual->proximo;  
    }  
    return false; // chegou ao fim sem encontrar  
}
```



Ex. 6 – Elementos comuns de duas listas ligadas

- Sobrecarga do operador – (usa ciclo while)

```
CListaInteiros CListaInteiros::operator- (CListaInteiros &l){  
    CListaInteiros aux; // lista resultado da operação  
    // Percorre a primeira lista...  
    CNoLista *noLista1 = cabeca;  
    while (noLista1 != nullptr) {  
        // para cada elemento verifica se existe na 2ª lista  
        if (l.procuraItem(noLista1->numero))  
            // se existir, inclui-o na lista resultado  
            aux.insereItem(noLista1->numero);  
  
        noLista1 = noLista1->proximo;  
    }  
    return aux; // devolve por cópia a nova lista  
}
```



Ex. 6 – Elementos comuns de duas listas ligadas

- Sobrecarga do operador – (usa ciclo for)

```
CListaInteiros CListaInteiros::operator- (CListaInteiros &l) {  
    CListaInteiros aux; // lista resultado da operação  
    // Percorre a primeira lista...  
    for (CNoLista *noLista1 = cabeca;  
         noLista1 != nullptr;  
         noLista1 = noLista1->proximo ) {  
        // para cada elemento verifica se existe na 2ª lista  
        if (l.procuraItem(noLista1->numero))  
            // se existir, inclui-o na lista resultado  
            aux.insereItem(noLista1->numero);  
    }  
    return aux; // devolve por cópia a nova lista  
}
```



Ex. 7 – Eliminar Penúltimo Elemento de uma Fila

- Definição de uma fila de números inteiros:

```
class CNoFila{  
    public:  
        int dados;  
        CNoFila *proximo;  
};  
  
class CFilaInteiros{  
    CNoFila *inicio;  
    CNoFila *fim;  
    public:  
        CFilaInteiros(void);  
        ~CFilaInteiros(void);  
  
        void insere(const int item);  
        bool retira(int &item);  
        ...  
};
```

- Implemente um novo método que permita eliminar o penúltimo elemento da fila, caso a fila tenha mais do que dois elementos.



Ex. 7 – Eliminar Penúltimo Elemento de uma Fila

- Caso de teste 1:
 - $L = 25 \rightarrow 46 \rightarrow 13 \rightarrow 26 \rightarrow 7 \rightarrow \text{nullptr}$
 - Fica: $L = 25 \rightarrow 46 \rightarrow 13 \rightarrow 7 \rightarrow \text{nullptr}$
- Caso de teste 2:
 - $L = 47 \rightarrow 13 \rightarrow \text{nullptr}$
 - O método não deve fazer nada.
- Caso de teste 3:
 - $L = 4 \rightarrow \text{nullptr}$
 - Também neste caso, o método não deve fazer nada.



Ex. 7 – Eliminar Penúltimo Elemento de uma Fila

```
void CFilaInteiros::eliminaPenultimo() {
    CNoFila *aux = inicio, *anterior;

    if(aux == nullptr) return;           // Fila tem 0 elementos
    if(aux->proximo == nullptr) return;  // Fila tem apenas 1 elemento
    if(aux->proximo->proximo == nullptr)
        return;                          // Fila tem apenas 2 elementos

    aux = aux->proximo; // Procura penúltimo a partir do 2º elemento
    while (aux->proximo != fim) {
        anterior = aux;
        aux = aux->proximo;
    } // No final do ciclo, aux aponta para penúltimo e
      // anterior aponta para antepenúltimo
    anterior->proximo = fim; // Antepenúltimo a apontar para último
    delete(aux);           // Elimina penúltimo
}
```



Ex. 8 – Prioridade a Pares numa Fila de Números

- Definição de uma fila de números inteiros (mesma que no ex. 7):

```
class CNoFila{
public:
    int dados;
    CNoFila *proximo;
};

class CFilaInteiros{
    CNoFila *inicio;
    CNoFila *fim;
public:
    CFilaInteiros(void);
    ~CFilaInteiros(void);

    void insere(const int item);
    bool retira(int &item);
    ...
};
```

- Implemente um novo método que dê prioridade na fila aos números pares, i.e. passe para o início os números pares.



Ex. 8 – Prioridade a Pares numa Fila de Números

- Caso de teste:

- $L = 4 \rightarrow 7 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 1 \rightarrow \text{nullptr}$

- Deve ter como resultado:

- $L = 4 \rightarrow 2 \rightarrow 6 \rightarrow 8 \rightarrow 7 \rightarrow 3 \rightarrow 5 \rightarrow 1 \rightarrow \text{nullptr}$

- Pretende-se uma solução que não utilize nenhuma estrutura de dados auxiliar.

- Numa fase intermédia (7 e 3 já passaram para o fim da lista):

- $L = 4 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 1 \rightarrow 7 \rightarrow 3 \rightarrow \text{nullptr}$



Ex. 8 – Prioridade a Pares numa Fila de Números

```
void CFilaInteiros::prioridadePares() {  
    CNoFila *atual,*anterior,*fimInicial,*seguinte;  
    bool chegueiAoFim; // true quando se chega ao fim da  
                        // fila original  
  
    if (inicio == nullptr) return; // retorna logo se fila  
vazia  
  
    atual = inicio;  
    fimInicial = fim; // Fim inicial da lista; poderá ser  
                    // diferente depois da ordenação  
  
    anterior = nullptr;  
    chegueiAoFim = false;  
    ...  
}
```



Ex. 8 – Prioridade a Pares numa Fila de Números

```
...
while (!chegueiAoFim){
    // Se atual aponta para elem. que estava no final da lista...
    if (atual == fimInicial) chegueiAoFim = true;
    if (atual->dados % 2 != 0){ // Se atual é ímpar vai para o fim
        seguinte = atual->proximo; // Guarda temp/ ponteiro p/ proximo
        // Vai retirar elemento ímpar (apontado por aux)...
        if (anterior == nullptr) // Se está no início da fila
            inicio = atual->proximo;
        else anterior->proximo = atual->proximo;
        // Insere elemento ímpar a seguir ao fim atual da fila...
        fim->proximo = atual;
        atual->proximo = nullptr; // passa a ser o último elemento
        fim = atual;              // idem
        atual = seguinte; //recupera ponteiro para elem a seguir na fila
    }
    ...
}
```



Ex. 8 – Prioridade a Pares numa Fila de Números

```
...  
    else { // Se o nó atual é par então passamos ao seguinte  
        anterior = atual;  
        atual = atual->proximo;  
    }  
}  
}
```